Linux 内核分析方法

2010-9-12

Linux 的最大的好处之一就是它的源码公开。同时，公开的核心源码也吸引着无数的电脑爱好者和程序员；他们把解读和分析 Linux 的核心源码作为自己的最大兴趣，把修改 Linux 源码和改造 Linux 系统作为自己对计算机技术追求的最大目标。

Linux 内核源码是很具吸引力的，特别是当你弄懂了一个分析了好久都没搞懂的问题；或者是被你修改过了的内核，顺利通过编译，一切运行正常的时候。那种成就感真是油然而生！而且，对内核的分析，除了出自对技术的狂热追求之外，这种令人生畏的劳动所带来的回报也是非常令人着迷的，这也正是它拥有众多追随者的主要原因：

- 首先，你可以从中学到很多的计算机的底层知识，如后面将讲到的系统的引导和硬件提供的中断机制等；其它，象虚拟存储的实现机制，多任务机制，系统保护机制等等，这些都是非读源码不能体会的。
- 同时，你还将从操作系统的整体结构中，体会整体设计在软件设计中的份量和作用，以及一些宏观设计的方法和技巧：Linux 的内核为上层应用提供一个与具体硬件不相关的平台；同时在内核内部，它又把代码分为与体系结构和硬件相关的部分，和可移植的部分；再例如，Linux 虽然不是微内核的，但他把大部分的设备驱动处理成相对独立的内核模块，这样减小了内核运行的开销，增强了内核代码的模块独立性。
- 而且你还能从对内核源码的分析中，体会到它在解决某个具体细节问题时，方法的巧妙：如后面将分析到了的 Linux 通过 Botoom_half 机制来加快系统对中断的处理。
- 最重要的是：在源码的分析过程中，你将会被一点一点地、潜移默化地专业化。一个专业的程序员，总是把代码的清晰性，兼容性，可移植性放在很重要的位置。他们总是通过定义大量的宏，来增强代码的清晰度和可读性，而又不增加编译后的代码长度和代码的运行效率；他们总是在编码的同时，就考虑到了以后的代码维护和升级。 甚至，只要分析百分之一的代码后，你就会深刻地体会到，什么样的代码才是一个专业的程序员写的，什么样的代码是一个业余爱好者写的。而这一点是任何没有真正分析过标准代码的人都无法体会到的。

然而，由于内核代码的冗长，和内核体系结构的庞杂，所以分析内核也是一个很艰难，很需要毅力的事；在缺乏指导和交流的情况下，尤其如此。只有方法正确，才能事半功倍。正是基于这种考虑，作者希望通过此文能给大家一些借鉴和启迪。

由于本人所进行的分析都是基于 2.2.5 版本的内核；所以，如果没有特别说明，以下分析都是基于 i386 单处理器的 2.2.5 版本的 Linux 内核。所有源文件均是相对于目录 /usr/src/linux 的。

方法之一：从何入手

要分析 Linux 内核源码，首先必须找到各个模块的位置，也即要弄懂源码的文件组织形式。虽然对于有经验的高手而言，这个不是很难；但对于很多初级的 Linux 爱好者，和那些对源码分析很有兴趣但接触不多的人来说，这还是很有必要的。

1、Linux 核心源程序通常都安装在/usr/src/linux 下，而且它有一个非常简单的编号约定：任何偶数的核心（的二个数为偶数，例如 2.0.30）都是一个稳定地发行的核心，而任何奇数的核心（例如 2.1.42）都是一个开发中的核心。

2、核心源程序的文件按树形结构进行组织，在源程序树的最上层，即目录/usr/src/linux 下有这样一些目录和文件：

◆ COPYING: GPL 版权申明。对具有 GPL 版权的源代码改动而形成的程序，或使用 GPL 工具产生的程序，具有使用 GPL 发表的义务，如公开源代码；

◆ CREDITS：光荣榜。对 Linux 做出过很大贡献的一些人的信息；

◆ MAINTAINERS：维护人员列表，对当前版本的内核各部分都有谁负责；

◆ Makefile：第一个 Makefile 文件。用来组织内核的各模块，记录了个模块间的相互这间的联系和依托关系，编译时使用；仔细阅读各子目录下的 Makefile 文件对弄清各个文件这间的联系和依托关系很有帮助；

◆ ReadMe：核心及其编译配置方法简单介绍；

◆ Rules.make：各种 Makefilemake 所使用的一些共同规则；

◆ REPORTING-BUGS:有关报告 Bug 的一些内容；

● Arch/ ：arch 子目录包括了所有和体系结构相关的核心代码。它的每一个子目录都代表一种支持的体系结构，例如 i386 就是关于 intel cpu 及与之相兼容体系结构的子目录。PC 机一般都基于此目录；

● Include/: include 子目录包括编译核心所需要的大部分头文件。与平台无关的头文件在 include/linux 子目录下，与 intel cpu 相关的头文件在 include/asm-i386 子目录下,而 include/scsi 目录则是有关 scsi 设备的头文件目录；

● Init/：这个目录包含核心的初始化代码(注：不是系统的引导代码)，包含两个文件 main.c 和 Version.c，这是研究核心如何工作的好的起点之一。

● Mm/：这个目录包括所有独立于 cpu 体系结构的内存管理代码，如页式存储管理内存的分配和释放等；而和体系结构相关的内存管理代码则位于 arch/*/mm/，例如 arch/i386/mm/Fault.c；

● Kernel/：主要的核心代码，此目录下的文件实现了大多数 linux 系统的内核函数，其中最重要的文件当属 sched.c；同样，和体系结构相关的代码在 arch/*/kernel 中；

● Drivers/：放置系统所有的设备驱动程序；每种驱动程序又各占用一个子目录：如，/block 下为块设备驱动程序，比如 ide（ide.c）。如果你希望查看所有可能包含文件系统的设备是如何初始化的，你可以看 drivers/block/genhd.c 中的 device_setup()。它不仅初始化硬盘，也初始化网络，因为安装 nfs 文件系统的时候需要网络；

● Documentation/: 文档目录,没有内核代码，只是一套有用的文档，可惜都是 English 的，看看应该有用的哦；

● Fs/: 所有的文件系统代码和各种类型的文件操作代码，它的每一个子目录支持一个文件系统, 例如 fat 和 ext2；

● Ipc/: 这个目录包含核心的进程间通讯的代码；

● Lib/: 放置核心的库代码；

● Net/: 核心与网络相关的代码；

● Modules/: 模块文件目录，是个空目录，用于存放编译时产生的模块目标文件；

● Scripts/: 描述文件，脚本，用于对核心的配置；

一般，在每个子目录下，都有一个 Makefile 和一个 Readme 文件，仔细阅读这两个文件，对内核源码的理解很有用。

对 Linux 内核源码的分析，有几个很好的入口点：一个就是系统的引导和初始化，即

从机器加电到系统核心的运行；另外一个就是系统调用，系统调用是用户程序或操作调用核心所提供的功能的接口。对于那些对硬件比较熟悉的爱好者，从系统的引导入手进行分析，可能来的容易一些；而从系统调用下口，则可能更合适于那些在 dos 或 Uinx、Linux 下有过 C 编程经验的高手。这两点，在后面还将介绍到。

方法之二：以程序流程为线索，一线串珠

从表面上看，Linux 的源码就象一团扎乱无章的乱麻，其实它是一个组织得有条有理的蛛网。要把整个结构分析清楚，除了找出线头，还得理顺各个部分之间的关系，有条不紊的一点一点的分析。

所谓以程序流程为线索、一线串珠，就是指根据程序的执行流程，把程序执行过程所涉及到的代码分析清楚。这种方法最典型的应用有两个：一是系统的初始化过程；二是应用程序的执行流程：从程序的装载，到运行，一直到程序的退出。

为了简便起见，遵从循序渐进的原理，现就系统的初始化过程来具体的介绍这种方法。系统的初始化流程包括：系统引导，实模式下的初始化，保护模式下的初始化共三个部分。下面将一一介绍。

inux 系统的常见引导方式有两种：Lilo 引导和 Loadin 引导；同时 **linux 内核**也自带了一个 bootsect-loader。由于它只能实现 linux 的引导，不像前两个那样具有很大的灵活性（lilo 可实现多重引导、loadin 可在 dos 下引导 linux）,所以在普通应用场合实际上很少使用 bootsect-loader。当然，bootsect-loader 也具有它自己的优点：短小没有多余的代码、附带在内核源码中、是内核源码的有机组成部分，等等。

bootsect-loader 在内和源码中对应的程序是 /Arch/i386/boot/bootsect.S 。下面将主要是针对此文件进行的分析。

1. 几个相关文件：

   <1> /Arch/i386/boot/bootsect.S

   <2> /include/linux/config.h

   <3> /include/asm/boot.h

   <4> /include/linux/autoconf.h
2. 引导过程分析：

对于 Intel x86 PC，开启电源后，机器就会开始执行 ROM BIOS 的一系列系统测试动作，包括检查 RAM，keyboard，显示器，软硬磁盘等等。执行完 bios 的系统测试之后，紧接着控制权会转移给 ROM 中的启动程序(ROM bootstrap routine)；这个程序会将磁盘上的第 0 轨第 0 扇区（叫 boot sector 或 MBR ，系统的引导程序就放在此处）读入内存中，并放到自 0x07C0:0x0000 开始的 512 个字节处；然后处理机将跳到此处开始执行这一引导程序；也即装入 MBR 中的引导程序后，CS:IP = 0x07C0:0x0000 。加电后处理机运行在与 8086 相兼容的实模式下。

如果要用 bootsect-loader 进行系统引导，则必须把 bootsect.S 编译连接后对应的二进制代码置于 MBR； 当 ROM BIOS 把 bootsect.S 编译连接后对应的二进制代码装入内存后，机器的控制权就完全转交给 bootsect； 也就是说，bootsect 将是第一个被读入内存中并执行的程序。

Bootsect 接管机器控制权后，将依次进行以下一些动作：

1．首先，bootsect 将它"自己"(自位置 0x07C0:0x0000 开始的 512 个字节)从被 ROM BIOS 载入的地址 0x07C0:0x0000 处搬到 0x9000:0000 处; 这一任务由 bootsect.S 的前十条指令完成；第十一条指令“jmpi go,INITSEG”则把机器跳转到“新”的 bootsect 的“jmpi go,INITSEG”后的那条指令“go: mov di,#0x4000-12”；之后，继续执行 bootsect 的剩下的代码；在 bootsect.S 中定义了几个常量：

BOOTSEG = 0x07C0 bios 载入 MBR 的约定位置的段址；

INITSEG = 0x9000 bootsect.S 的前十条指令将自己搬到此处(段址)

SETUPSEG =0x9020 装入 Setup.S 的段址

SYSSEG =0x1000 系统区段址

对于这些常量可参见/include/asm/boot.h 中的定义；这些常量在下面的分析中将会经常用到；

2．以 0x9000:0x4000-12 为栈底，建立自己的栈区；其中 0x9000:0x4000-12 到 0x9000:0x4000 的一十二个字节预留作磁盘参数表区；

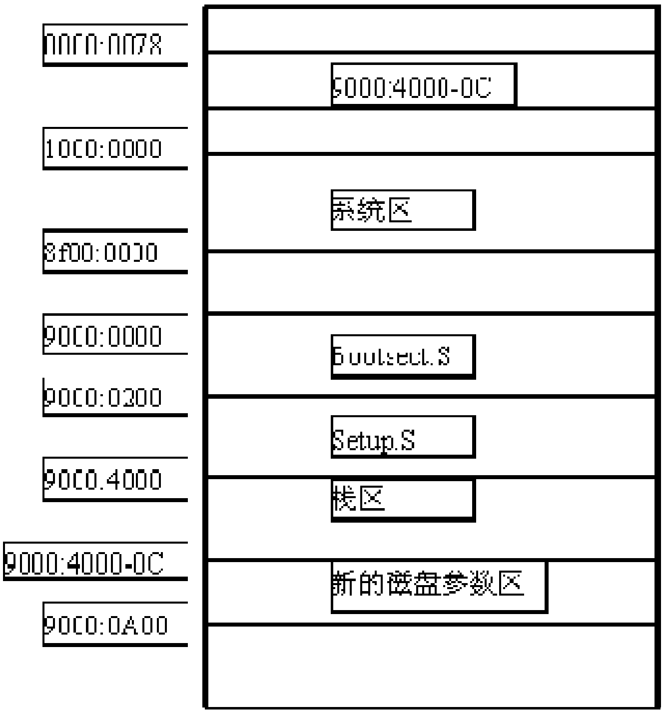
3．在 0x9000:0x4000-12 到 0x9000:0x4000 的一十二个预留字节中建立新的磁盘参数表，之所以叫“新”的磁盘参数表，是相对于 bios 建立的磁盘参数表而言的。由于设计者考虑到有些老的 bios 不能准确地识别磁盘“每个磁道的扇区数”，从而导致 bios 建立的磁盘参数表妨碍磁盘的最高性能发挥，所以，设计者就在 bios 建立的磁盘参数表的基础上通过枚举法测试，试图建立准确的“新”的磁盘参数表(这是在后继步骤中完成的)；并把参数表的位置由原来的 0x0000:0x0078 搬到 0x9000:0x4000-12；且修改老的磁盘参数表区使之指向新的磁盘参数表；

4．接下来就到了 load_setup 子过程；它调用 0x13 中断的第 2 号服务；把第 0 道第 2 扇区开始的连续的 setup_sects (为常量 4)个扇区读到紧邻 bootsect 的内存区；，即 0x9000:0x0200 开始的 2048 个字节；而这四个扇区的内容即是 /arch/i386/boot/setup.S 编译连接后对应的二进制代码；　也就是说，如果要用 bootsect-loader 进行系统引导，不仅必须把 bootsect.S 编译连接后对应的二进制代码置于 MBR，而且还得把 setup.S 编译连接后对应的二进制代码置于紧跟 MBR 后的连续的四个扇区中；当然，由于 setup.S 对应的可执行码是由 bootsect 装载的，所以，在我们的这个项目中可以通过修改 bootsect 来根据需要随意地放置 setup.S 对应的可执行码；

5．load_setup 子过程的唯一出口是 probe_loop 子过程；该过程通过枚举法测试磁盘"每个磁道的扇区数";

6.接下来几个子过程比较清晰易懂:打印我们熟悉的"Loading"；读入系统到 0x1000:0x0000; 关掉软驱马达；根据的 5 步测出的"每个磁道的扇区数"确定磁盘类型；最后跳转到 0x9000:0x0200,即 setup.S 对应的可执行码的入口，将机器控制权转交 setup.S;整个 bootsect 代码运行完毕；

3．引导过程执行完后的内存印象图：



出于简便考虑，在此分析中，我忽略了对大内核的处理的分析，因为对大内核的处理，只是此引导过程中的一个很小的部分，并不影响对整体的把握。完成了系统的引导后，系统将进入到初始化处理阶段。系统的初始化分为实模式和保护模式两部分。

II、实模式下的初始化

实模式下的初始化,主要是指从内核引导成功后，到进入保护模式之前系统所做的一些处理。在内核源码中对应的程序是 /Arch/i386/boot/setup.S；以下部分主要是针对此文件进行的分析。这部分的分析主要是要弄懂它的处理流程和 INITSEG(9000:0000)段参数表的建立，此参数表包含了很多硬件参数，这些都是以后进行保护模式下初始化，以及核心建立的基础。

1. 几个其它相关文件： <1> /Arch/i386/boot/bootsect.S

<2> /include/linux/config.h

<3> /include/asm/boot.h

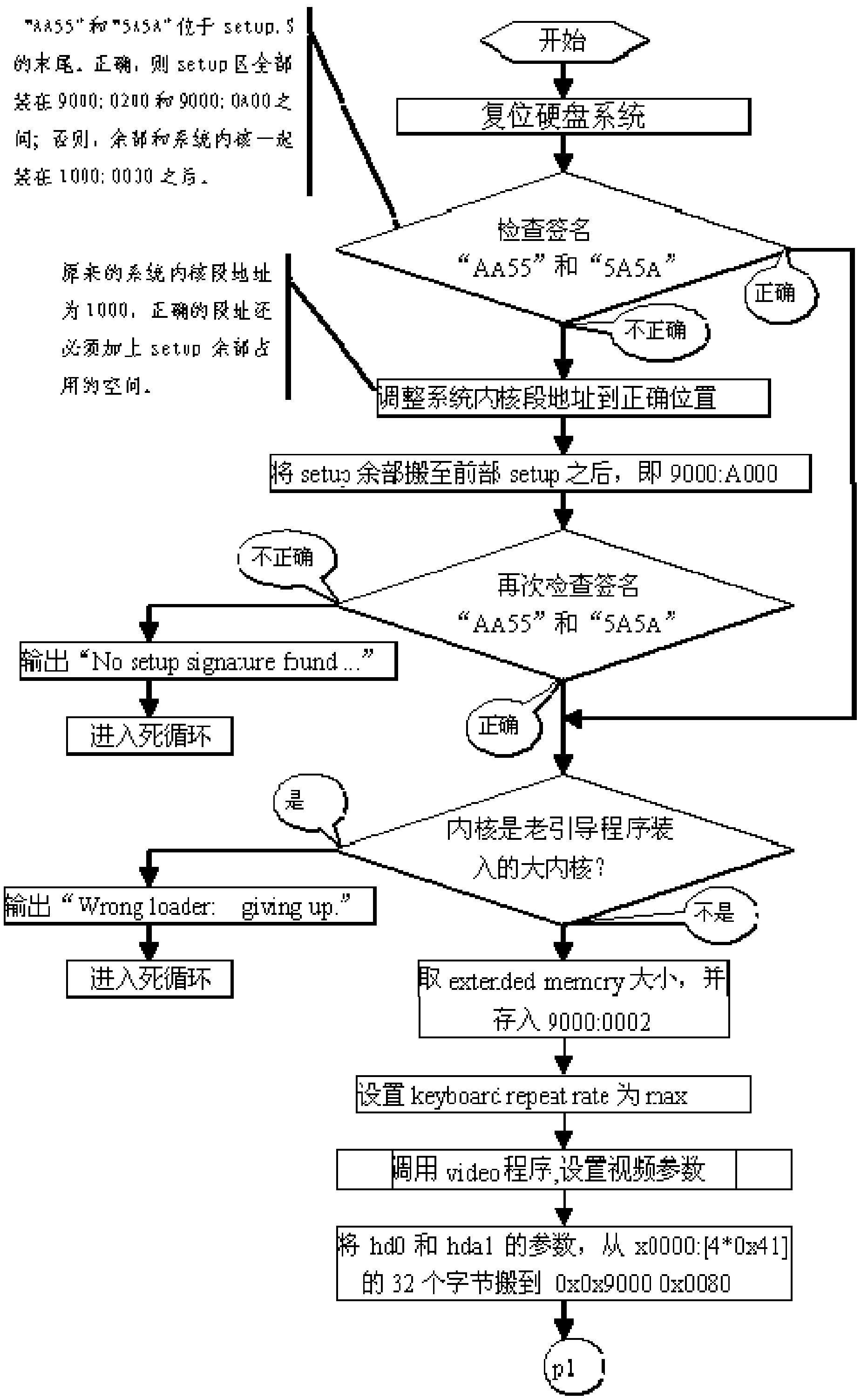
<4> /include/ asm/segment.h

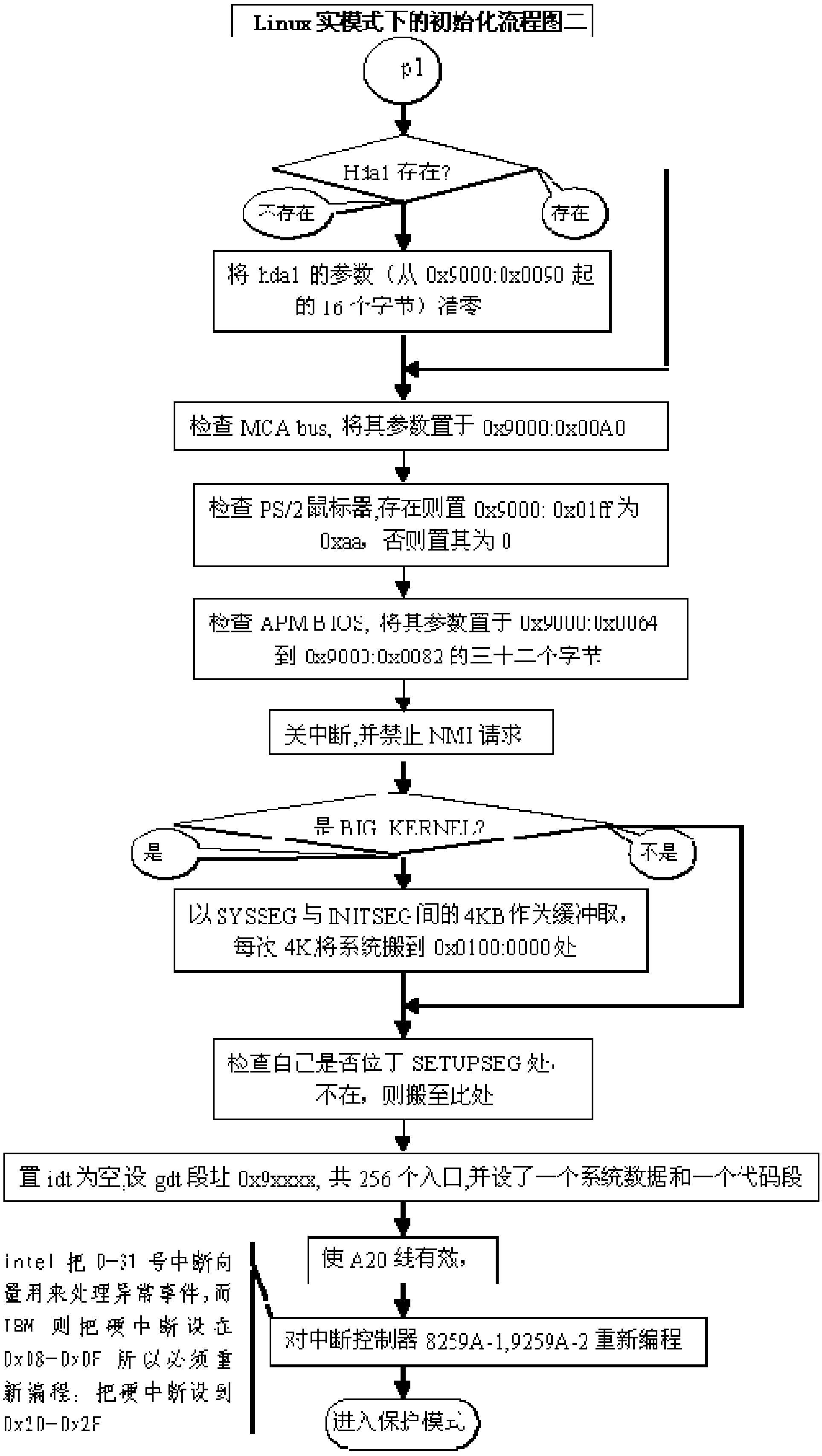
<5> /include/linux/version.h

<6> /include/linux/compile.h

2. 实模式下的初始化过程分析：

Linux实模式下的初始化流程图一





INITSEG(9000:0000)段参数表：（参见 **Include/linux/tty.h**）

参数名

偏移量(段址均为 **0x9000)**

长度 **Byte**

参考文件

PARAM_CURSOR_POS

0x0000

2

Arch/i386/boot/video.S

extended mem Size

0x0002

2

Arch/i386/boot/setup.S

PARAM_VIDEO_PAGE

0x0004

2

Arch/i386/boot/video.S

PARAM_VIDEO_MODE

0x0006

1

Arch/i386/boot/video.S

PARAM_VIDEO_COLS

0x0007

1

Arch/i386/boot/video.S

没用

0x0008

2

Include/linux/tty.h

PARAM_VIDEO_EGA_BX

0x000a

2

Arch/i386/boot/video.S

没用

0x000c

2

Include/linux/tty.h

PARAM_VIDEO_LINES

0x000e

1

Arch/i386/boot/video.S

PARAM_HAVE_VGA

0x000f

1

Arch/i386/boot/video.S

PARAM_FONT_POINTS

0x0010

2

Arch/i386/boot/video.S

PARAM_LFB_WIDTH

0x0012

2

Arch/i386/boot/video.S

PARAM_LFB_HEIGHT

0x0014

2

Arch/i386/boot/video.S

PARAM_LFB_DEPTH

0x0016

2

Arch/i386/boot/video.S

PARAM_LFB_BASE

0x0018

4

Arch/i386/boot/video.S

PARAM_LFB_SIZE

0x001c

4

Arch/i386/boot/video.S

暂未用①

0x0020

4

Include/linux/tty.h

PARAM_LFB_LINELENGTH

0x0024

2

Arch/i386/boot/video.S

PARAM_LFB_COLORS

0x0026

6

Arch/i386/boot/video.S

暂未用②

0x002c

2

Arch/i386/boot/video.S

PARAM_VESAPM_SEG

0x002e

2

Arch/i386/boot/video.S

PARAM_VESAPM_OFF

0x0030

2

Arch/i386/boot/video.S

PARAM_LFB_PAGES

0x0032

2

Arch/i386/boot/video.S

保留

0x0034--0x003f

Include/linux/tty.h

APM BIOS Version③

0x0040

2

Arch/i386/boot/setup.S

BIOS code segment

0x0042

2

Arch/i386/boot/setup.S

BIOS entry offset

0x0044

4

Arch/i386/boot/setup.S

BIOS 16 bit code seg

0x0048

2

Arch/i386/boot/setup.S

BIOS data segment

0x004a

2

Arch/i386/boot/setup.S

支持 32 位标志④

0x004c

2

Arch/i386/boot/setup.S

BIOS code seg length

0x004e

4

Arch/i386/boot/setup.S

BIOS data seg length

0x0052

2

Arch/i386/boot/setup.S

hd0 参数

0x0080

16

Arch/i386/boot/setup.S

hd0 参数

0x0090

16

Arch/i386/boot/setup.S

PS/2 device 标志⑤

0x01ff

1

Arch/i386/boot/setup.S

* 注： ① Include/linux/tty.h ： CL_MAGIC and CL_OFFSET here

1. Include/linux/tty.h ：
unsigned char rsvd_size; /* 0x2c */ unsigned char rsvd_pos; /* 0x2d */

③ 0 表示没有 APM BIOS

④ 0x0002 置位表示支持 32 位模式

⑤ 0 表示没有，0x0aa 表示有鼠标器

III、保护模式下的初始化

保护模式下的初始化，是指处理机进入保护模式后到运行系统第一个内核程序过程中系统所做的一些处理。保护模式下的初始化在内核源码中对应的程序是 /Arch/i386/boot/compressed/head.S 和 /Arch/i386/KERNEL/head.S ；以下部分主要是针对这两个文件进行的分析。

1. 几个相关文件：

<1.> /Arch/i386/boot/compressed/head.S

<2.> /Arch/i386/KERNEL/head.S

<3.> //Arch/i386/boot/compressed/MISC.c

<4.> /Arch/i386/boot/setup.S

<5.> /include/ asm/segment.h

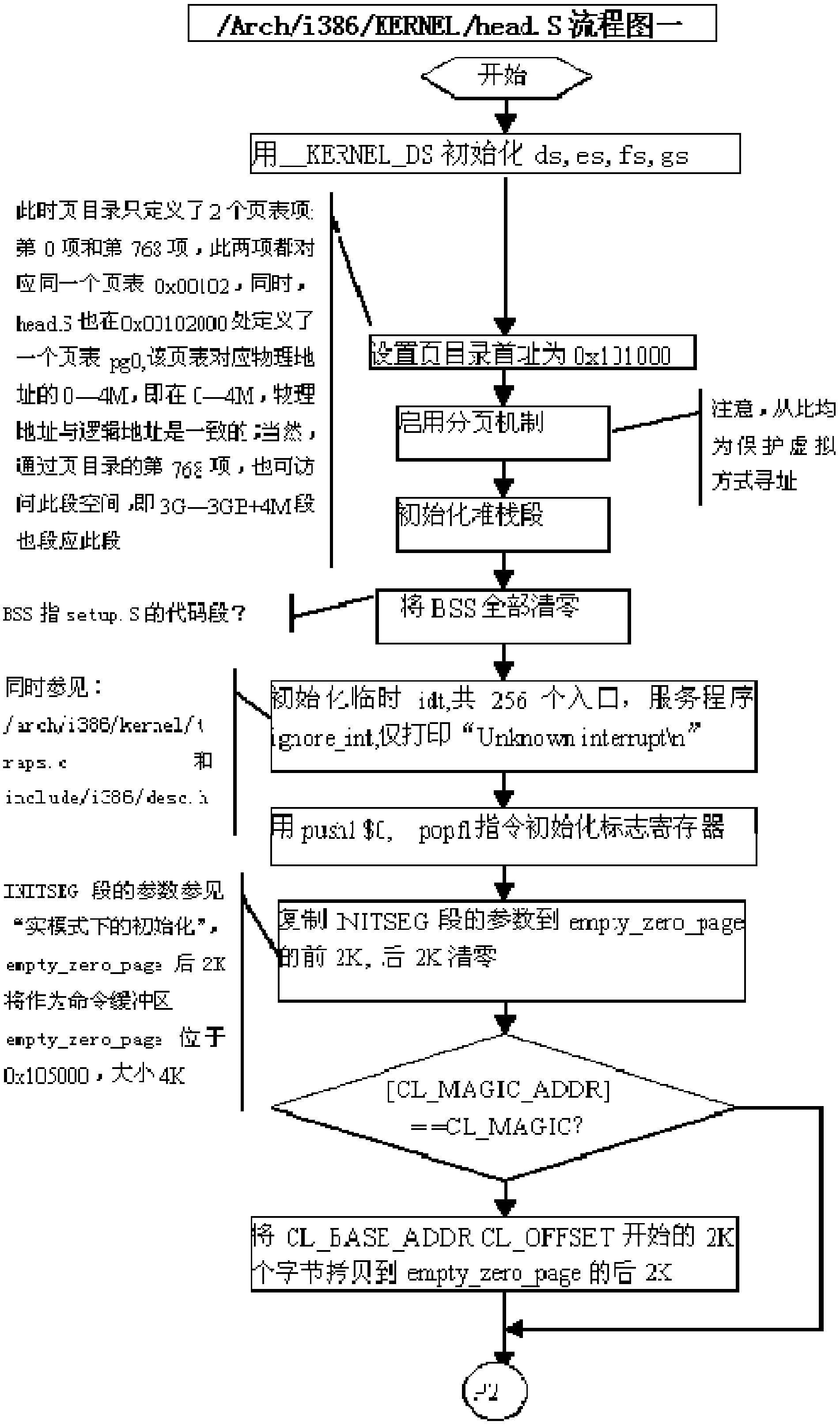
<6.> /arch/i386/kernel/traps.c

<7.> /include/i386/desc.h

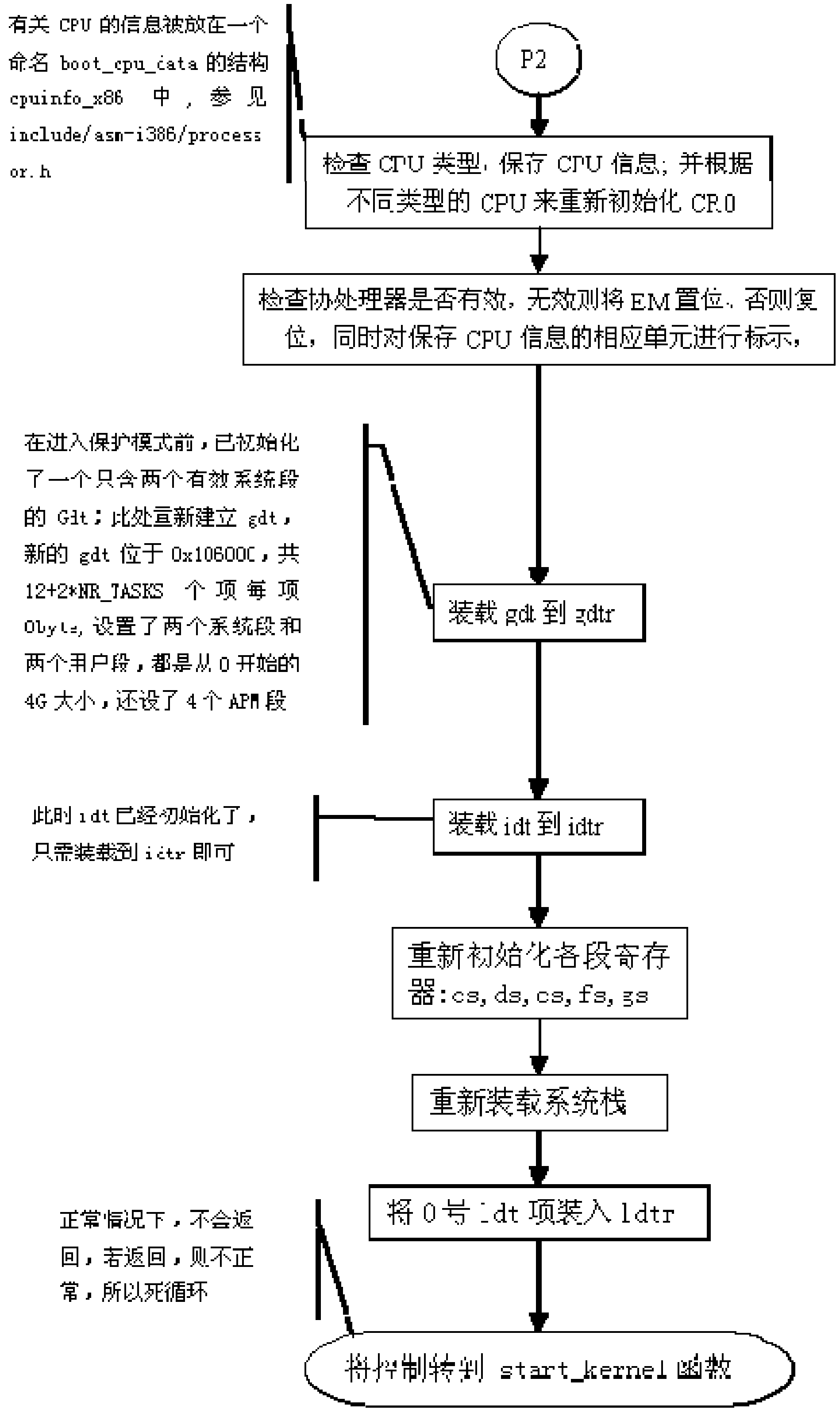
<8.> /include/asm-i386/processor.h

2. 保护模式下的初始化过程分析：

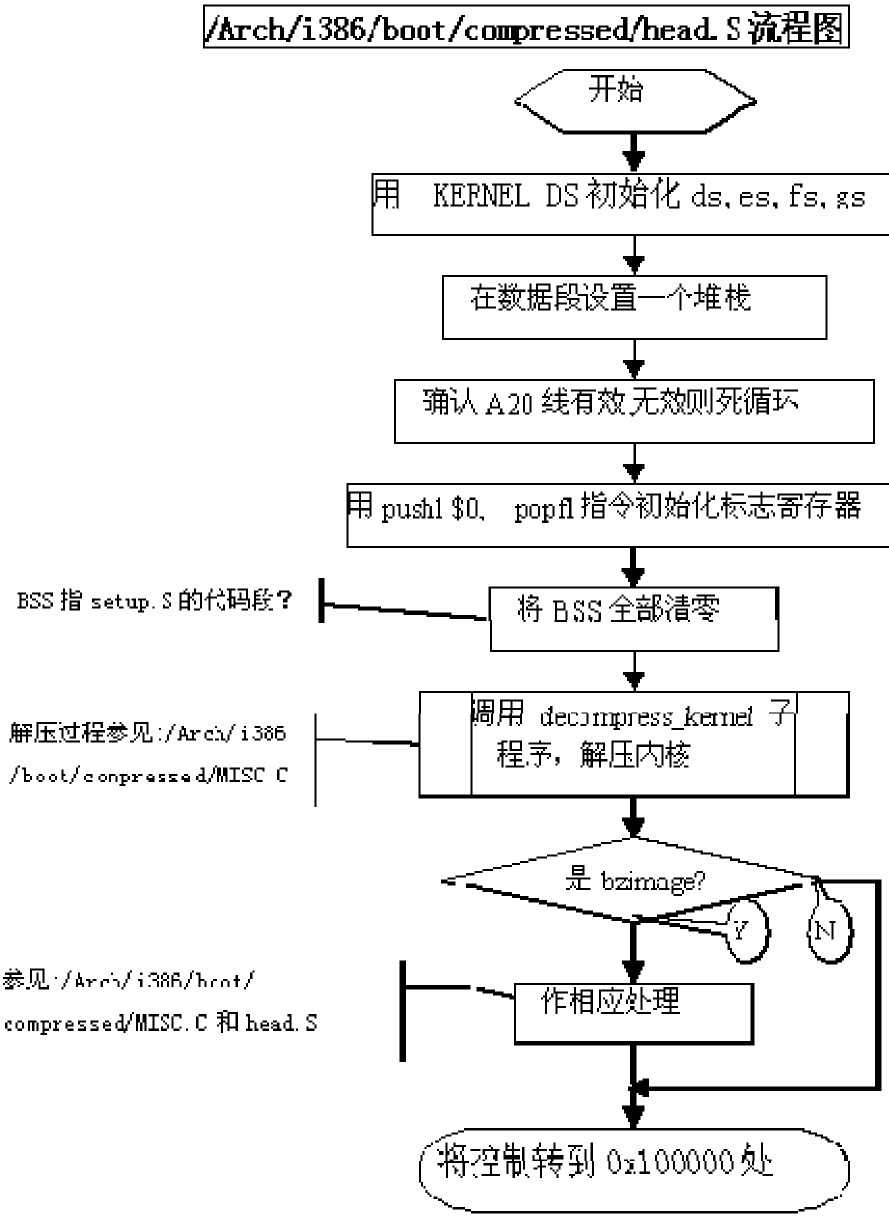
一、**/Arch/i386/KERNEL/head.S** 流程：



/Arch/i386/KERNEL/head.S 流程图二

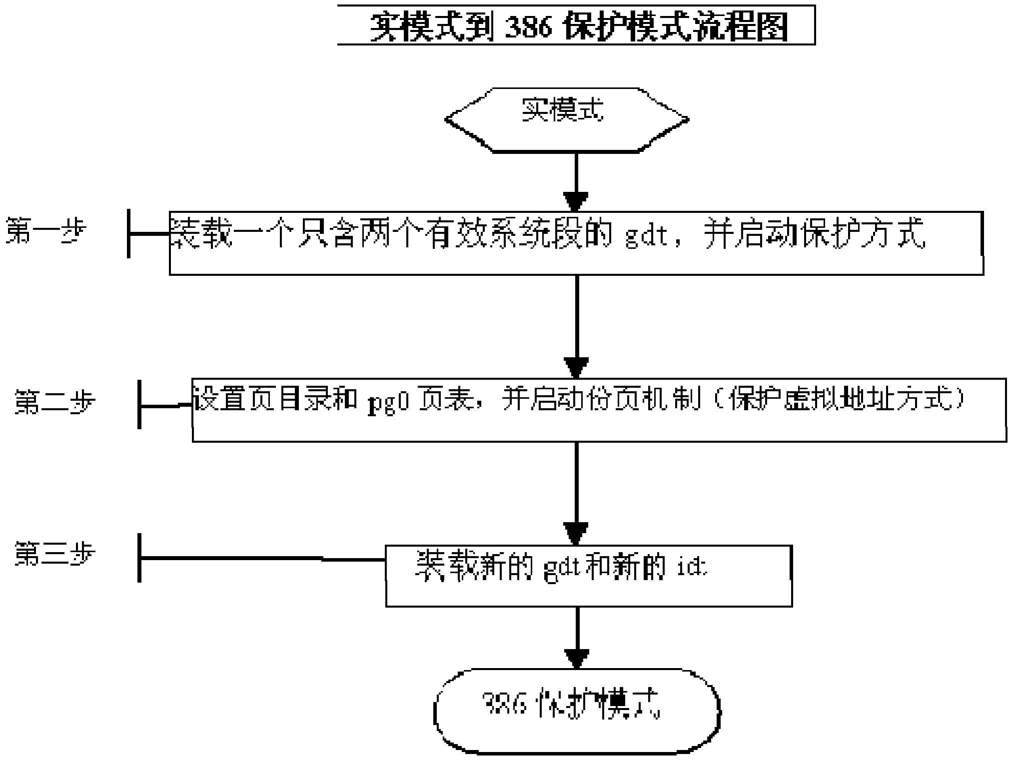


二、/Arch/i386/boot/compressed/head.S 流程：



1. 从流程图中可以看到，保护模式下的初始化主要干了这样几件事：
    a. 解压内核到 0x100000 处、
    b. 建立页目录和 pg0 页表并启动分页功能(即虚存管理功能)、
    c. 保存实模式下测到的硬件信息到 empty_zero_page、初始化命令缓存区、
    d. 检测 cpu 类型、检查协处理器、
    e. 重新建立 gdt 全局描述符表、和中断描述附表 idt；

2. 从页目录和 pg0 页表可以看出，0&#0；4M 物理内存被用作系统区，它被映射到系统段线性空间的 0&#0；4M 和 3G&#0；3G+4M；即系统可以通过访问这两个段来访问实际的 0&#0；4M 物理内存，也就是系统所在的区域；
3. 本来在实模式下初始化时已经建立了全局描述符表 gdt,而此处重新建立全局描述符表 gdt 则主要是出于两个原因：一个就是若内核是大内核 bzimag，则以前建立的 gdt，可能已经在解压时被覆盖掉了所以，在这个源码文件中均只采用相对转移指令 jxx nf 或 jxx nb；二是以前建立的 gdt 是建立在实地址方式下的，而现在则是在启用保护虚拟地址方式之后建立的，也即现在的 gdt 是建立在逻辑地址（即线性地址）上的；
4. 每次建立新的 gdt 后和启用保护虚拟地址方式后都必须重新装载系统栈和重新初始化各段寄存器:cs,ds,es,fs,gs；
5. 从实模式下的初始化和保护模式下的初始化过程可以看出，linux 系统由实模式进入到保护模式的过程大致如下：



6. 由于分页机制只能在保护模式下启动，不能在实模式下启动，所以第一步是必要的；又因为在 386 保护模式下 gdt 和 idt 是建立在逻辑地址(线性地址)上的，所以第三步也是必要的；

7. 经过实模式和保护模式下的初始后，主要系统数据分布如下：

初始后主要系统数据分布表

位置

系统数据

大小

0x101000

页目录 swapper_pg_dir

4K

0x102000

页表 pg0

4K

0x103000

empty_bad_page

4K

0x104000

empty_bad_page_table

4K

0x105000

empty_zero_page

4K

0x105000

系统硬件参数

2K

0x105800

命令缓冲区

2K

0x106000

全局描述附表 gdt_table

4192B

从上面对 Linux 系统的初始化过程的分析可以看出，以程序执行流程为线索、一线串珠，就是按照程序的执行先后顺序，弄懂程序执行的各个阶段所进行的处理，及其各阶段之间的相互联系。而流程图应该是这种分析方法最合适的表达工具。

事实上，以程序执行流程为线索，是分析任何源代码都首选的方法。由于操作系统的特殊性，光用这种方法是远远不够的。当然用这种方法来分析系统的初始化过程或用户进程的执行流程应该说是很有效的。